Observation temps-réel de programmes Caml

Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant, Julien Robert, Guillaume von Tokarski

Observation temps-réel de programmes Caml

S. Conchon[1,2,3] & J.-C. Filliâtre[2,1,3] & F. Le Fessant[3] & J. Robert[1] & G. Von Tokarski[1]

1: Université Paris Sud F-91405 Orsay
2: CNRS / LRI UMR 8623 F-91405 Orsay
3: INRIA Saclay – Île-de-France F-91893 Orsay
{conchon,filliatr,jrobert,gvt}@lri.fr, fabrice.le_fessant@inria.fr

Résumé

Pour mettre au point un programme, tant du point de vue de sa correction que de ses performances, il est naturel de chercher à observer son exécution. On peut ainsi chercher à observer la gestion de la mémoire, le temps passé dans une certaine partie du code, ou encore certaines valeurs calculées par le programme. De nombreux outils permettent de telles observations (moniteur système, *profiler* ou *debugger* génériques ou spécifiques au langage, instrumentation explicite du code, etc.). Ces outils ne proposent cependant que des analyses « après coup » ou des observations très limitées. Cet article présente Ocamlviz, une bibliothèque pour instrumenter du code OCaml et des outils pour visualiser ensuite son exécution, en temps-réel et de manière distante.

1. Introduction

Il existe de nombreux moyens pour mettre au point un programme écrit en Objective Caml [3]. S'il s'agit de sa correction, la solution la plus simple consiste souvent à insérer quelques affichages sur la sortie standard, par exemple à l'aide de la fonction `printf`. Si cette trace ne suffit pas, on peut se tourner vers le *debugger* d'OCaml [4], qui permet notamment d'inspecter toute valeur calculée et même de revenir en arrière dans l'exécution du programme. Il est également possible d'utiliser un debugger plus générique tel que `gdb` [1]. S'il s'agit d'analyser les performances du programme, il existe des solutions simples et immédiates, comme les outils Unix `time` et `top` pour mesurer respectivement le temps d'exécution et la quantité de mémoire utilisée. Pour une analyse plus fine, on peut utiliser des *profilers* tels que celui fourni avec OCaml, `ocamlprof` [5], ou encore des outils génériques comme `gprof` [2] et `OProfile` [6].

Même s'ils sont complémentaires, ces divers outils possèdent un grand nombre de limitations. La première concerne les possibilités d'observation. Les profilers génériques `gprof` et `OProfile` sont ainsi limités à l'observation du nombre d'appels de chaque fonction et du temps passé dans celles-ci. Inversement, `ocamlprof` a une granularité plus fine, mais ne donne que des décomptes de points de passage sans indication de temps de calcul. Les debuggers non plus ne permettent pas de mesurer le temps de calcul. D'autre part, aucun de ces outils ne permet de mesurer la quantité de mémoire occupée par une valeur OCaml. Concernant l'aspect temps-réel, seul `OProfile` permet d'observer un programme en cours d'exécution. Des outils comme `ocamlprof` et `gprof` n'offrent qu'une analyse *post-mortem*. Enfin, aucun effort particulier n'est fait dans ces outils pour présenter graphiquement les résultats des observations. Leur architecture ne permet généralement pas de leur ajouter facilement une telle fonctionnalité.

Le projet Ocamlviz propose une alternative à ces différents outils, sous la forme d'une bibliothèque pour instrumenter du code OCaml et des outils pour visualiser ensuite son exécution, en temps-réel et de manière distante. Les fonctions proposées dans cette bibliothèque permettent au programmeur

d'indiquer précisément les observations qu'il souhaite faire. En particulier, la granularité de ces observations n'est pas limitée aux fonctions. L'utilisateur peut ainsi observer des points de passage et des temps d'exécution pour des portions de code arbitraires. D'autre part, il est possible de mesurer la quantité de mémoire utilisée par une ou plusieurs valeurs et même de déterminer si elles sont encore vivantes pour le ramasse-miettes.

Une particularité d'Ocamlviz est de dissocier le calcul des valeurs observées et leur exploitation. En effet, le programme instrumenté à l'aide de la bibliothèque se comporte, pendant son exécution, comme un serveur auquel des clients peuvent se connecter pour récupérer les résultats de l'observation. Ocamlviz fournit un client particulier, sous la forme d'une interface graphique GTK-2. L'architecture d'Ocamlviz peut être ainsi illustrée : Le serveur et les clients communiquent à l'aide d'un protocole

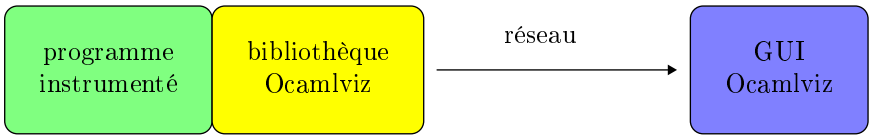


FIG. 1 – Architecture d'Ocamlviz.

réseau indépendant d'OCaml, ce qui permet d'observer l'exécution depuis des machines distantes et d'écrire des clients dans d'autres langages.

Cet article est organisé de la manière suivante. La section 2 décrit l'architecture générale d'Ocamlviz et montre les principales fonctionnalités d'instrumentation fournies par la bibliothèque. La section 3 décrit les détails importants de l'implantation de la bibliothèque et du protocole réseau. Enfin, la section 4 présente un certain nombre de perspectives pour Ocamlviz.

2. Principe

Ocamlviz est à la fois une bibliothèque pour instrumenter du code OCaml à observer et des outils pour transmettre et visualiser les résultats. Le principe de fonctionnement est illustré figure **??**. En premier lieu, l'utilisateur instrumente son code en indiquant les observations qu'il souhaite réaliser. Il utilise pour cela les briques de base fournies par la bibliothèque Ocamlviz. Il lie ensuite son programme avec cette bibliothèque. Le code ainsi obtenu peut être exécuté normalement mais il peut également être observé. En effet, en parallèle de l'exécution normale, le programme se comporte maintenant comme un serveur qui attend des connections et transmet le résultat des observations à ses clients. En particulier, Ocamlviz fournit une interface graphique évoluée permettant une visualisation agréable et synthétique des résultats (voir figure 2).

Le caractère client/serveur de l'architecture offre de nombreux avantages. Premièrement, il permet de décider d'observer le programme à n'importe quel moment de son exécution. En particulier, un client peut se connecter longtemps après le début de l'exécution et se déconnecter à tout instant. Deuxièmement, un nombre arbitraire de clients peuvent se connecter à un même programme en cours d'exécution. Troisièmement, les clients peuvent se connecter depuis des machines distantes, qu'elles aient ou non la même architecture que celle sur laquelle s'exécute le code instrumenté. Enfin, cette architecture dissocie l'instrumentation de l'observation des résultats. En particulier, le protocole de communication est indépendant du langage OCaml et permet par exemple l'utilisation de clients écrits dans d'autres langages.

L'instrumentation minimale consiste simplement à lier son programme avec la bibliothèque Ocamlviz. Cela a pour effet immédiat d'envoyer aux clients des données relatives au GC d'OCaml (taille totale du tas et taille de sa partie vivante). L'interface graphique d'Ocamlviz se charge alors

FIG. 2 – L'interface graphique d'Ocamlviz.

d'afficher cette information sous forme d'un graphe, tel qu'on peut le voir sur la figure 2. Au delà de cette première information, la bibliothèque Ocamlviz offre les possibilités suivantes :

- mesurer le temps passé entre deux points du programme ;
- compter le nombre de passages en un ou plusieurs points du programme ;
- observer des valeurs calculées par le programme ;
- mesurer le nombre et la taille totale d'un ensemble de valeurs désignées par l'utilisateur ;
- afficher des messages à la `printf` qui sont archivés dans un journal.

Le reste de cette section détaille d'une part ces différentes instrumentations et d'autre part les fonctionnalités de l'interface graphique d'Ocamlviz.

2.1. Instrumentation

La bibliothèque Ocamlviz contient plusieurs modules pour observer des temps de calcul, des points de passage, ou encore des valeurs calculées par le programme.

Mesure du temps. Ocamlviz fournit une notion de chronomètres, que l'on peut déclencher et arrêter en tout point du programme. Ces points de programme ne coïncident pas nécessairement avec le début et la fin d'une fonction, comme c'est le cas généralement dans les outils de *profiling*. D'autre part, on peut mesurer, par accumulation, le temps passé dans plusieurs sections du programme. On crée un chronomètre de la manière suivante :

```
let chrono = Ocamlviz.Time.create "t"
```

La chaîne de caractères `"t"` n'est utile que pour l'affichage dans l'interface graphique. Le chronomètre s'utilise alors ainsi :

```
let f x =
  ...
  Ocamlviz.Time.start chrono;
  let z = ... in
  Ocamlviz.Time.stop chrono
  ...
```

FIG. 3 – Chronomètres et points de programme.

Dans cet exemple, on mesure le temps passé dans le calcul de `z` mais pas dans le reste de la fonction `f`. Il est important de noter que le chronomètre peut être déclenché dans une fonction et arrêté dans une autre. La figure 3 montre comment l'interface graphique présente la valeur du chronomètre `"t"` (ici 49,9 secondes représentant 100% du temps total d'exécution).

Points de programme. Ocamlviz fournit un moyen de marquer un ou plusieurs points de programme puis de compter le nombre de fois que l'exécution passe par ces marqueurs. On crée un tel marqueur de la manière suivante :

```
let point = Ocamlviz.Point.create "p"
```

On marque alors les points de programme que l'on souhaite observer avec

```
...
Ocamlviz.Point.observe point;
...
```

Un même marqueur peut être utilisé à plusieurs endroits du programme et le décompte est global à chaque marqueur. La figure 3 montre ainsi que l'on est passé 803 800 206 fois par des points de programme marqués avec `point`.

Observer des valeurs. Ocamlviz permet d'observer des valeurs arbitraires calculées par le programme. Pour les types simples (entiers, flottants ou chaînes de caractères), il suffit de fournir une fonction calculant la valeur à observer. Dès lors, cette fonction est évaluée régulièrement, à une fréquence que l'utilisateur peut spécifier. Par exemple, le code suivant

```
let () = Ocamlviz.Value.observe_float_fct
           "my value" ~period:200 (fun () → sin !v)
```

déclare une fonction d'observation calculant la valeur flottante `sin !v` toutes les 200 millisecondes. La figure 4 montre la visualisation de cette valeur dans l'interface graphique.

Pour les types plus complexes, une solution simple consiste à transformer la valeur à observer en chaîne de caractères. Cependant, Ocamlviz fournit un moyen plus élégant d'observer des valeurs telles que des arbres ou des graphes. Pour cela, l'utilisateur commence par transformer sa valeur dans un type de la forme suivante :

```
type t = { node : string; mutable children : t list }
```

Il s'agit donc, en toute généralité, d'un type de graphes dont les nœuds sont étiquetés par des chaînes de caractères. L'aspect `mutable` du champ `children` permet en effet de construire des valeurs cycliques. Si la donnée de type `t` construite par l'utilisateur contient du partage, celui-ci sera préservé par le protocole de communication et présenté dans l'interface graphique fournie par Ocamlviz. La figure 5 illustre la visualisation d'une valeur structurée où le nœud étiqueté `"shared"` est partagé.

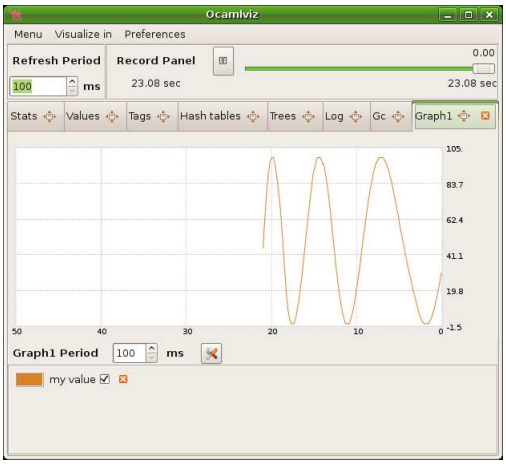


FIG. 4 – Observation de valeur scalaire.

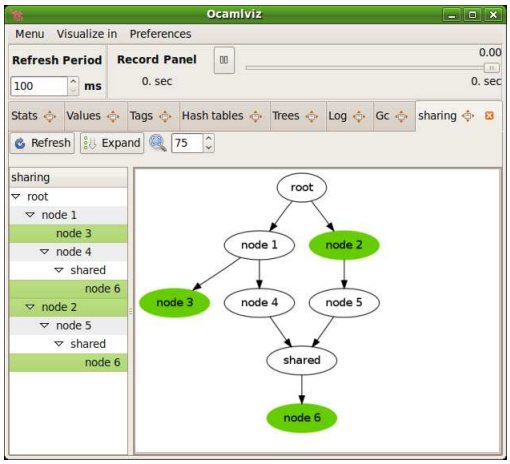


FIG. 5 – Observation d'une valeur structurée.

Marquer des valeurs. Ocamlviz permet d'analyser l'utilisation de la mémoire plus finement qu'à travers les informations globales fournies par le GC. Ocamlviz donne en effet la possibilité de marquer des valeurs puis de connaître, à tout instant, le nombre de ces valeurs toujours vivantes et l'espace mémoire qu'elles occupent.

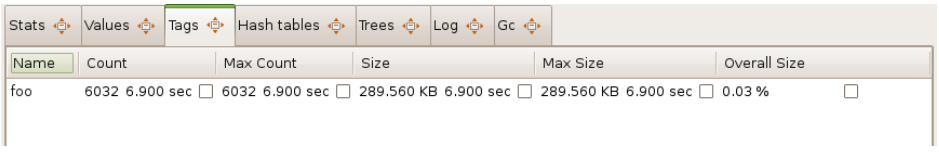
Comme pour les points de programme, on commence par créer un marqueur :

```
let t = Ocamlviz.Tag.create ~size:true ~period:300 "foo"
```

On crée ici un marqueur de nom `"foo"`. On spécifie que l'on souhaite calculer la taille occupée (option `size`) et la calculer toutes les 300 millisecondes (option `period`). On peut par exemple marquer une valeur qui vient d'être construite, comme dans la fonction suivante :

```
let cons x =
  let l = Random.float 10. :: x in
  Ocamlviz.Tag.mark t l;
  l
```

L'interface graphique permet de visualiser en temps réel le nombre et la taille de chaque marqueur.



Sur cette capture, on lit que 6 032 valeurs encore vivantes sont marquées avec le tag `"foo"` et qu'elles occupent un peu plus de 289 ko.

Un même marqueur peut être utilisé pour marquer une ou plusieurs valeurs, qu'elles soient ou non du même type. Si plusieurs données sont marquées avec le même marqueur, et qu'elles partagent des valeurs, alors les données partagées ne sont comptées qu'une seule fois dans le calcul de l'espace mémoire occupé. Ainsi dans le code suivant

```
let l2 = [Random.int 3; Random.int 4] in
Ocamlviz.Tag.mark t l2;
```

```
let l3 = Random.int 5 :: l2 in
Ocamlviz.Tag.mark t l3;
...
```

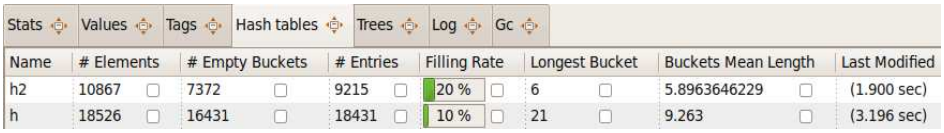
le nombre de valeurs marquées par le tag `t` est 2 (les listes `l2` et `l3` et la taille mémoire occupée pour ce tag est de 36 octets (3 blocs *cons*, de 3 mots chacun en comptant l'entête de bloc).

Tables de hachage. L'observation d'une valeur d'un type abstrait introduit dans la bibliothèque standard d'OCaml est relativement difficile : il faut d'une part connaître la définition du type et d'autre part utiliser le module `Obj`. Par exemple, l'utilisateur ne peut pas observer facilement le contenu d'une table de hachage de type `Hashtbl.t`, en particulier pour déterminer si la répartition dans les différents *buckets* est bien uniforme. Pour remédier à cela, Ocamlviz fournit un module pour observer une table de hachage de la bibliothèque standard.

Ainsi, la ligne suivante déclare une table de hachage globale, dont on souhaite observer le contenu sous le nom `"h"` :

```
let h = Ocamlviz.Hashtable.observe "h" (Hashtbl.create 17)
```

Dès lors, l'interface graphique affiche plusieurs éléments concernant cette table :

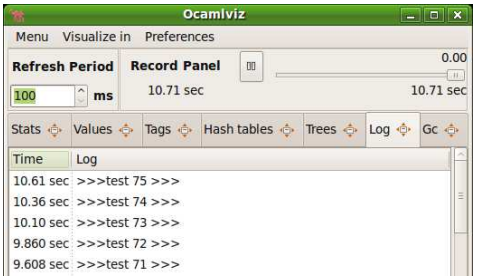


On trouve en particulier le nombre de *buckets* vides, le taux de remplissage du tableau, c'est-à-dire la proportion de *buckets* non vides, ou encore la longueur du plus grand *bucket*.

Journal. Enfin, Ocamlviz fournit une facilité « à la `printf` » pour transmettre des messages jusqu'aux clients. Ces messages sont transmis avec l'indication du temps écoulé depuis le début de l'exécution du programme. On peut ainsi écrire

```
...
Ocamlviz.log ">>> test %d >>>" i;
```

et observer dans l'interface la succession des messages transmis :



Instrumentation automatique avec camlp4. Il peut s'avérer fastidieux d'instrumenter un code de manière systématique, par exemple pour observer le temps passé dans *chaque* fonction, à la manière d'un profiler traditionnel. Ocamlviz fournit pour cela un petit module `camlp4` qui insère automatiquement les observations suivantes :

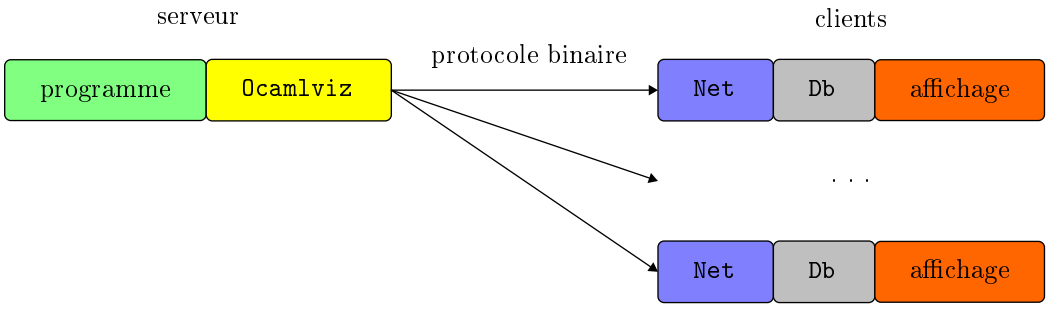


FIG. 6 – Ocamlviz est composé d'une bibliothèque liée au programme instrumenté, qui communique via le réseau avec un ou plusieurs clients.

- valeur de toute référence globale de type `int`, `float`, `bool` ou `string` (initialisée par une constante) ;
- contenu de toute table de hachage globale ;
- nombre d'appels et temps passé dans chaque fonction globale.

Ce module `camlp4` a surtout pour objectif de montrer qu'une instrumentation automatique est possible ; il est probable que chaque utilisateur le modifiera pour ses propres besoins.

2.2. Autres aspects de l'interface graphique

Au delà de celles présentées ci-dessus, l'interface graphique fournie avec Ocamlviz propose également d'autres fonctionnalités. En particulier, l'utilisateur peut interrompre à tout moment le rafraîchissement de l'interface, le temps d'observer tranquillement les informations transmises par le serveur. Le programme en cours d'exécution n'est pas interrompu et il continue en particulier d'émettre des données. Celles-ci ne sont pas perdues mais sont accumulées par le client dans une base de données, qui sera décrite plus loin (section 3.4). À tout moment, l'utilisateur peut reprendre l'observation du programme. Celle-ci reprend exactement là où elle a été interrompue et donc avec un décalage dans le temps (d'une manière analogue à la fonctionnalité de `time shifting` d'un magnétoscope numérique). Plus généralement, l'utilisateur peut se déplacer dans le temps d'une manière arbitraire (à l'aide d'un curseur), dans la limite d'une fenêtre d'enregistrement de taille prédéfinie (mais réglable). Cette fonctionnalité apparaît dans la partie supérieure de l'interface graphique, sous la forme suivante :

Une autre fonctionnalité de l'interface est la possibilité de visualiser une ou plusieurs valeurs numériques sous forme de graphes, à l'instar de ce qui est fait pour les données du ramasse-miettes. Pour cela, il suffit de sélectionner la ou les valeurs à visualiser (en les cochant) puis de demander la création d'un nouveau graphe ou même l'ajout à un graphe existant, à des fins de comparaison. Les graphes ainsi obtenus apparaissent dans de nouveaux onglets mais peuvent être détachés de l'interface afin que plusieurs graphes puissent être observés en même temps.

3. Réalisation

Comme illustré sur la Figure 6, Ocamlviz se décompose en une bibliothèque serveur liée au programme instrumenté, qui communique via le réseau avec un ou plusieurs clients, et une bibliothèque qui permet d'écrire aisément des clients.

Dans cette section, nous décrivons comment le serveur et les clients ont été implantés dans la version courante d'Ocamlviz. Plus précisément, on décrit d'abord le protocole de communication et sa réalisation, puis certains aspects du calcul des observations et enfin le module de stockage des données côté client.

3.1. Protocole de communication

Une capacité importante d'Ocamlviz est la possibilité d'observer le fonctionnement d'un programme s'exécutant sur une machine depuis une autre machine. Pour permettre une hétérogénéité maximale entre ces deux machines, un protocole binaire et portable a été spécifié et implanté dans les bibliothèques serveur et client. Ceci permet d'une part la communication entre des machines d'architectures différentes (32 et 64 bits par exemple, mais aussi *little-endian* et *big-endian*[1], mais aussi l'écriture de clients et de serveurs dans d'autres langages qu'OCaml. Ainsi, il sera possible dans l'avenir d'écrire des bibliothèques serveurs permettant d'instrumenter des programmes dans d'autres langages qu'OCaml et de les observer depuis le client graphique actuel d'Ocamlviz, mais aussi d'écrire de nouveaux clients dans d'autres langages.

3.1.1. Messages

Le protocole actuel ne contient que des messages du serveur vers les clients. Il se compose de trois messages :

```
type msg =
  | Declare of uid × kind × string
  | Send of uid × value
  | Bind of uid list
```

Les messages désignent des valeurs observées, qui sont identifiées par des entiers uniques de type `uid`. Le message `Declare (uid, kind, name)` déclare au client une nouvelle valeur observée, en indiquant son identifiant, sa nature de type `kind` et le nom qui sera utilisé pour l'affichage. Au moment de sa connexion, le client reçoit du serveur un ensemble de messages `Declare` correspondant à toutes les valeurs observées dans le programme instrumenté jusqu'à cet instant.

Le message `Send (uid, v)` fournit une mise à jour de la valeur courante de l'identifiant `uid` avec la valeur `v`. Il est envoyé régulièrement pour chaque valeur, même si celle-ci n'a pas changé entre-temps.

Enfin, le message `Bind uid_list` indique au client qu'un certain nombre d'identifiants sont liés entre eux ; c'est en particulier le cas pour les deux valeurs correspondant au nombre et la taille d'un marqueur de type `Ocamlviz.Tag.t`.

3.1.2. Protocole binaire

Chaque message est une chaîne de caractères : elle se décompose d'un premier entête de 4 octets, indiquant la longueur totale de la chaîne, d'un second entête indiquant le type du message, puis enfin des arguments du message, dont l'ordre et le type dépendent du type du message.

[1] Les termes anglais *little-endian* et *big-endian* ont été empruntés aux *Voyages de Gulliver* de Jonathan Swift. Il serait donc naturel de les traduire en français par « petits-boutien » et « gros-boutien ».

Cette représentation permet d'effectuer une analyse efficace de chaque message. La connaissance de sa longueur dès l'entête permet notamment de ne commencer l'analyse que lorsque tout le message a été lu. Elle permet aussi de n'allouer une chaîne de caractères pour la lecture du message que si la taille de celui-ci dépasse la taille de la chaîne par défaut (65000 actuellement), évitant les allocations de petites chaînes de caractères qui fragmentent le tas et ralentissent le fonctionnement du ramasse-miettes.

Pour faciliter l'écriture et l'extension du protocole, des fonctions sont définies pour transmettre chaque type OCaml de base, puis combinées pour transmettre les types plus complexes. Pour chaque type de base *ttype* (entiers 8, 16, 32 ou 64 bits, flottant, etc.) on introduit le couple de fonctions suivant :

```
val get_ttype : string → int → ttype × int
val buf_ttype : Buffer.t → ttype → unit
```

La fonction `get_`*ttype* `s pos` extrait une valeur de type *ttype* de la chaîne `s` à partir de la position `pos` et renvoie un couple contenant cette valeur et la position suivante dans la chaîne. La fonction `buf_`*ttype* `buf v` encode la valeur `v` de type *ttype* à la fin du tampon `buf`.

Pour insérer la taille de chaque message, quatre octects nuls sont placés en tête du tampon avant l'encodage du message. La chaîne correspondant au message est ensuite extraite du tampon puis, sa longueur étant connue, ses quatre premiers octects sont modifiés en conséquence.

Enfin, pour permettre une compatibilité entre architectures 32 et 64 bits, les valeurs manipulées par le client portent une marque de type, en particulier indiquant pour les entiers s'ils sont sur 32 (pour les `int` 31 bits et les `int32`) ou 64 bits (pour les `int` 63 bits et les `int64`).

3.2. Bibliothèque serveur

Le bibliothèque serveur calcule toutes les 100 millisecondes l'ensemble des valeurs en cours d'observation et les envoie à tous les clients connectés. Elle gère aussi les connexions de nouveaux clients. Le délai entre les observations peut être modifié en utilisant une variable d'environement `OCAMLVIZ_PERIOD`.

Deux mécanismes — les alarmes et les processus légers (*threads*) — ont été implantés pour effectuer ces opérations régulières, suivant les contraintes liées au programme à observer.

3.2.1. Les alarmes

Les alarmes permettent de déclencher l'exécution d'une fonction à intervalles de temps réguliers. Cette technique fonctionne bien dans la plupart des cas, car elle interrompre l'exécution du programme complètement et ne souffre donc pas de problème de synchronisation. Il existe néanmoins certains cas où les alarmes peuvent poser problème :

- Les alarmes ne peuvent pas interrompre le programme à n'importe quel instant. En effet, OCaml ne permet l'exécution du code associé à une alarme qu'à certains moments particuliers, afin d'éviter les interactions avec le ramasse-miettes. En particulier, OCaml ne permet la gestion des alarmes qu'au moment des allocations et des entrées-sorties. Aussi, un programme qui ne fait que calculer, sans allocation ni entrée-sortie, ne verra jamais ses alarmes traitées et le serveur Ocamlviz n'enverra aucune donnée. Pour remédier à ce problème, il est possible d'insérer dans le code un appel `Ocamlviz.yield ()` qui donne l'opportunité à OCaml de traiter les alarmes, le cas échéant.
- Quand le programme utilise déjà des alarmes, il devient impossible pour Ocamlviz de les utiliser, car il n'y qu'un seul gestionnaire associé aux alarmes. Il devient alors nécessaire d'utiliser les processus légers pour ne pas modifier la sémantique du programme d'origine.

3.2.2. Les processus légers

Cette deuxième solution consiste à lancer au démarrage du programme un processus léger, dont la seule tâche est d'appeler régulièrement la fonction d'observation des valeurs. Cependant, cette solution comporte également des inconvénients :

- L'utilisation de processus légers a certaines limites en OCaml, en particulier parce que le ramasse-miettes n'est pas concurrent. En conséquence, la plateforme d'exécution ne permet l'exécution de code OCaml que d'un seul processus léger à la fois.
- Comme pour les alarmes, l'ordonnancement des processus légers ne s'effectue pas à n'importe quel instant, mais uniquement lors des allocations et des entrées-sorties. Là encore, un appel explicite à `Ocamlviz.yield ()` peut être nécessaire pour permettre au processus léger d'observation de s'exécuter un court instant.

3.3. Techniques d'observation du serveur

Les techniques d'observation mises en œuvre dans Ocamlviz tendent à perturber le moins possible l'exécution du programme. En particulier, afin de ne pas empêcher la collecte des valeurs mortes par le ramasse-miettes, les données marquées par l'utilisateur avec `Ocamlviz.Tag.mark` sont stockées dans des tables de pointeurs faibles (`weak pointers`). Une solution immédiate consiste à utiliser le module `Weak` de la bibliothèque standard d'OCaml de la manière suivante :

```
module WeakHash = Weak.Make(struct
  type t = Obj.t
  let hash = Hashtbl.hash
  let equal = (==)
end)
```

Cependant, cette solution s'avère à la fois incorrecte et inefficace. Elle est inefficace car la fonction générique de hachage `Hashtbl.hash` peut avoir un coût important lorsqu'elle est appliquée à des valeurs de taille importante. Si par exemple l'utilisateur choisit de marquer tous les éléments d'une liste, alors le seul coût de l'insertion dans la table de pointeurs faibles peut devenir quadratique. Une solution consiste à remplacer `Hashtbl.hash` par une fonction de coût constant qui limite la descente récursive dans les valeurs OCaml. Nous avons donc été contraints d'écrire une fonction de hachage dédiée (la fonction `Hashtbl.hash_param` peut sembler être faite pour cela, mais sa notion de bloc significatif — les types de base — est exactement l'opposée de celle qui est requise ici — les pointeurs).

La solution ci-dessus est également incorrecte. En effet, l'utilisation de la fonction (==) comme test d'égalité ne permet pas de comparer les valeurs ajoutées dans des tables de pointeurs faibles . On peut montrer ce défaut à l'aide de la séquences d'instructions suivantes :

```
# let t = WeakHash.create 17;;
val t : WeakHash.t = <abstr>
# let v = Obj.repr [1];;
val v : Obj.t = <abstr>
#  WeakHash.mem t v;;
_ : bool = false
```

La cause de ce problème est que `WeakHash.mem t v` ne compare pas `v` à chaque valeur `w` adressée par les pointeurs faibles de `t`, mais à des «copies» de celles-ci. En effet, une comparaison directe de `v` avec `w` créerait une racine temporaire qui pourrait retarder la collecte de `w` par le ramasse-miettes. Pour être plus précis, `WeakHash.mem` crée un copie de `w` en dupliquant son premier bloc et en partageant les

pointeurs vers sa structure interne. Ainsi, pour obtenir une solution correcte, il nous a suffit d'utiliser une fonction d'égalité comparant, par un test physique, les pointeurs contenus dans les premiers blocs de `v` et de la copie de `w` :

```
let copy_equal x y =
  if Obj.is_block x && Obj.is_block y &&
    Obj.size x = Obj.size y then
      let len = Obj.size x in
      let rec loop i x y len =
        (i = len) ||
          (Obj.field x i == Obj.field y i && loop (i+1) x y len)
      in
      loop 0 x y len
  else false
```

Le calcul de la taille des valeurs marquées se fait alors en profondeur en prenant soin de traiter correctement les données partagées ou cycliques (on utilise pour cela une table de hachage créée de la même manière que nos tables de pointeurs faibles).

Bien que correcte et relativement efficace, notre solution n'en reste pas moins coûteuse pour des données de taille importante ou lorsque la fréquence d'observation des valeurs marquées est élevée. Enfin, il est également important de noter que le compilateur OCaml réalise automatiquement un partage des données statiques identiques (dans le segment de données). Elles ne seront alors décomptées qu'une seule fois par Ocamlviz, ce qui peut donner des résultats surprenants.

3.4. Techniques de stockage du client

Pour facilier l'écriture des clients, Ocamlviz fournit plusieurs modules OCaml. Sur la figure 6 page 7 apparaissent notamment les deux modules `Net` et `Db`. Le premier assure la connection avec le serveur puis lit et décode les messages en provenance de celui-ci. Le second enregistre les données reçues dans une base de données, dans la limite d'une fenêtre de temps qui peut être spécifiée. Il permet alors d'interroger le contenu de cette base de données en se plaçant à un moment précis du temps d'exécution du programme. Ainsi la fonctionnalité de « magnétoscope numérique » évoquée page 7 est fournie gratuitement à tout client construit au dessus du module `Db`.

La base de données est construite au dessus d'une structure de dictionnaires indexés par le temps. La signature de cette structure est la suivante :

```
type α t
val create : ?size:int → α → α t
val add : α t → float → α → unit
val find : α t → float → float × α
val remove_before : α t → float → unit
```

Le type α `t` est le type d'une structure impérative associant à des valeurs flottantes, représentant ici des temps d'exécution, des données quelconques de type α. La fonction `create` crée un nouveau dictionnaire, contenant une unique donnée associée au temps 0. Il est possible de spécifier une capacité initiale, à titre indicatif, sans que cela soit cependant nécessaire. L'appel à `add` d t x ajoute dans le dictionnaire d l'association d'une donnée x au temps t, en supposant que t est supérieur ou égal à toutes les clés déjà présentes dans d. L'appel à `find` d t permet alors de retrouver dans d la donnée la plus récente (t', x), avec $t' \leq t$. Enfin, la fonction `remove_before` permet de supprimer dans un dictionnaire toutes les entrées antérieures à un certain temps.

Il est clair qu'une telle structure de données peut être directement exploitée pour réaliser le module `Db`. Chaque donnée observée se voit associée à un tel dictionnaire, qui est rempli au fur et à mesure que les données arrivent. (Le temps d'exécution est transmis par ailleurs, comme toute autre valeur observée.) L'hypothèse que fait la fonction `add` n'est pas contraignante car les données sont de fait stockées avec des temps croissants. La fonction `remove_before` est utilisée pour supprimer les données qui sortent de la fenêtre d'enregistrement, afin de ne pas effondrer le client sous la masse des données transmises.

Il est moins évident, en revanche, de déterminer comment réaliser une telle structure de données. Les structures de données fournies dans la bibliothèque standard d'OCaml n'offrent pas de solution immédiate. On pourrait songer à utiliser un arbre binaire de recherche, car y trouver la plus grande valeur inférieure ou égale à une clé donnée n'est pas vraiment plus difficile que la fonction traditionnelle de recherche. En revanche il est plus difficile de réaliser la fonction `remove_before` car on peut être amené à rééquilibrer une grande partie de l'arbre. Enfin l'insertion est logarithmique, ce qui n'exploite pas vraiment le fait que les données arrivent par ordre croissant de temps.

Nous avons finalement opté pour la solution suivante. Les éléments du dictionnaire sont stockés linéairement dans un tableau (en fait deux tableaux, un pour les temps et l'autre pour les éléments), qui est utilisé circulairement. L'insertion se fait donc en temps constant. Lorsque le tableau est plein, on le redimensionne en doublant sa taille (mais l'insertion reste donc de coût *amorti* constant). La fonction `find` utilise une recherche dichotomique (*binary search*) et son coût est donc logarithmique. Il faut tenir compte de l'utilisation circulaire du tableau, mais c'est là la seule difficulté. Enfin la fonction `remove_before` réutilise la fonction de recherche dichotomique pour déterminer l'index du plus jeune élément à supprimer ; un simple décalage de l'indice du premier élément suffit alors à supprimer d'un seul coup tous les éléments plus anciens. Le coût de `remove_before` est donc également logarithmique.

4. Conclusion et perspectives

Le projet Ocamlviz est un logiciel libre[2] qui a été financé par Jane Street Capital dans le cadre du programme *Jane Street Summer Project*. L'idée a été proposée par les trois premiers auteurs de cet article et réalisée par Julien Robert et Guillaume Von Tokarski, deux étudiants de M1 de l'Université Paris Sud. En tant que projet d'été, Ocamlviz semble avoir donné satisfaction :

> « From my point of view, the single most useful project is unquestionably ocamlviz. Ocamlviz is a realtime profiling tool for OCaml, and I was really impressed with the system's polish. The design is carefully thought out ; it seems to be quite well implemented ; the front-end has a surprisingly usable UI ; there's a nice looking website for it, and good documentation to boot. It's really a fantastic effort, and I expect we'll be taking it for a spin on some of our own OCaml projects. »
>
> Yaron Minsky, Jane Street Capital [8]

Par ailleurs, les premières expériences menées avec Ocamlviz ont été très concluantes. En particulier, Ocamlviz a été utilisé pour analyser un programme OCaml réaliste, à savoir le démonstrateur automatique Alt-Ergo [7]. Il a permis notamment, avec une instrumentation légère et ciblée, d'identifier précisément les causes de non-terminaison du démonstrateur sur certains exemples. Ceci était clairement impossible à réaliser avec des outils tels que des debuggers ou des profilers, et très fastidieux à l'aide de traces d'exécution « à la `printf` ».

Il est clair qu'Ocamlviz est naturellement adapté à de telles observations ciblées. Sa bibliothèque a été conçue pour permettre à l'utilisateur d'instrumenter spécifiquement certaines parties de son code, et donc d'en affecter très peu l'exécution. En particulier, les paramètres de certaines fonctions de la bibliothèque permettent un réglage fin des nuisances de l'observation. En revanche, il est peu évident

[2] Ocamlviz est disponible à l'adresse `http://ocamlviz.forge.ocamlcore.org/`.

que l'instrumentation globale de tout un programme donne des résultats exploitables. En particulier, l'exécution risque fort d'être très perturbée par les calculs effectués par le serveur.

Plusieurs aspects d'Ocamlviz peuvent être améliorés. D'une part, certaines informations manquent. Ainsi le temps passé dans le ramasse-miettes n'est pas disponible. Même s'il s'agit d'une information que l'on peut obtenir indirectement avec un outil comme `gprof`, il serait plus intéressant d'en disposer directement dans les informations fournies par le serveur. Malheureusement, cela nécessiterait de modifier le moteur d'exécution (*runtime*) d'OCaml, ce qui a été exclus pour garantir la pérennité d'Ocamlviz.

Concernant l'interface graphique, le mécanisme de pause s'avère très pratique pour étudier tranquillement les informations fournies par le programme. En revanche, il manque la possibilité d'indiquer un « point d'arrêt » dans le programme source qui aurait pour effet de stopper l'interface graphique à cet endroit précis de l'exécution (en revanche, l'exécution elle-même ne serait pas stoppée). Il ne s'agirait donc pas d'un point d'arrêt au sens d'un debugger. Techniquement, il s'agit d'une extension relativement simple : il suffit de faire passer par le protocole une valeur particulière, interprétée par l'interface graphique comme un ordre de pause.

Plus généralement, le protocole pourrait être étendu pour permettre également de faire transiter des informations des clients vers le serveur. Cela permettrait par exemple de modifier certains paramètres de l'observation : ne plus observer une valeur ; changer la fréquence à laquelle elle est observée ; etc.

Remerciements. Les auteurs, étudiants comme encadrants, remercient la société Jane Street Capital pour avoir financé le projet Ocamlviz et l'INRIA Saclay–Île-de-France pour avoir accueilli les étudiants.

Références

[1] `gdb` — The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[2] `gprof` — The GNU Profiler. `http://sourceware.org/binutils/docs-2.19/gprof/index.html`.

[3] Le langage Objective Caml. `http://caml.inria.fr/`.

[4] `ocamldebug` — The Objective Caml Debugger. .

[5] `ocamlprof` — The Objective Caml Profiler. `http://caml.inria.fr/pub/docs/manual-ocaml/manual031.html`.

[6] OProfile — A System Profile for Linux. `http://oprofile.sourceforge.net`.

[7] Sylvain Conchon et Evelyne Contejean. Alt-Ergo, un démonstrateur automatique dédié à la preuve de programmes. `http://alt-ergo.lri.fr/`.

[8] Yaron Minsky. Jane Street Summer Project round-up. `http://janestcapital.com/?q=node/68`.